ETH Zürich
Chair of Software Engineering

C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 – May 2007

Lecture 9: Threads ...

Lisa (Ling) Liu

Overview

Thread state

WaitHandle Synchronization mechanism

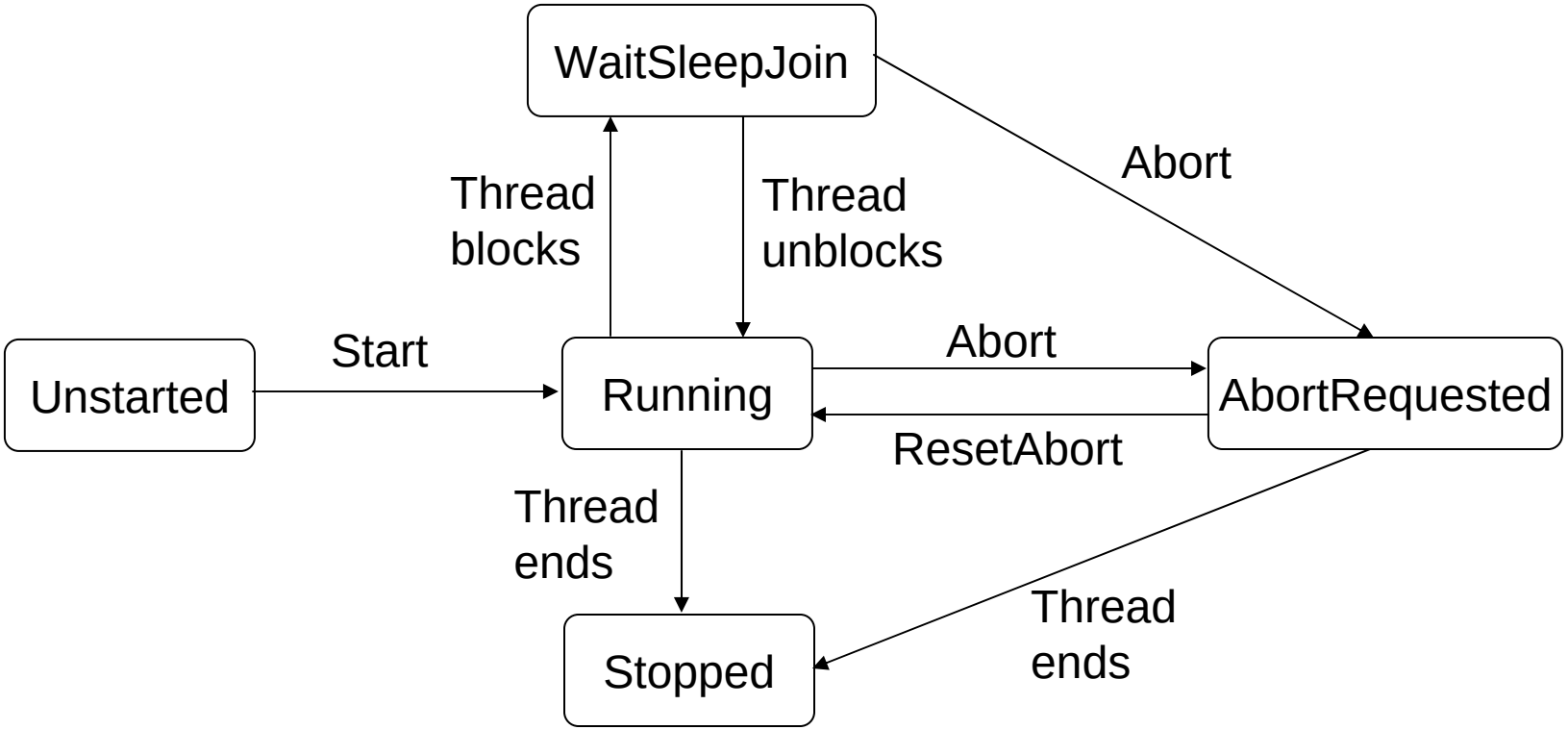
Synchronize access to collections

Automatic synchronization

Atomicity and interlocked

Using ThreadPool

Thread state



Blocking

- Once a thread blocks, it immediately relinquishes its resource, enters WaitSleepJoin state and doesn't get re-scheduled until unblocked.
- Four unblocking ways
  - By the blocking condition being satisfied
  - By the operation timing out (if a timeout is specified)
  - By being interrupted via Thread.Interrupt
  - By being aborted via Thread.Abort

Interrupt and abort

Interrupt

- Throw a ThreadInterruptedException
- Called on a non-blocking thread doesn't affect the execution of the thread

Abort

- Throw a ThreadAbortException
- Rethrow the exception at the end of the catch block unless Thread.ResetAbort is called
- Called on a non-blocking thread causes an exception

Suspend and resume a thread

thread.Suspend()

- Temporarily suspends a running thread
- A thread can suspend itself

thread.Resume()

- Restarts a suspended thread

Blocking synchronization

Synchronized code regions (SyncBlock based)

- lock, Monitor

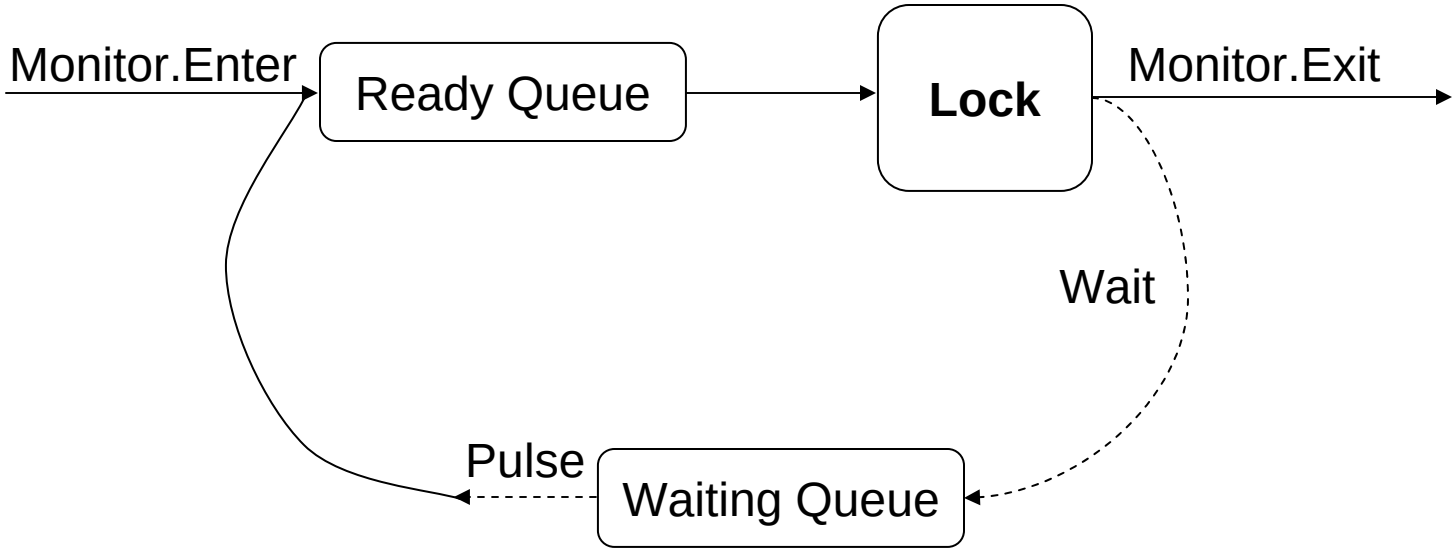
Classic manual synchronization

- WaitHandle, Mutex, ReadWriterLock, ManualResetEvent, AutoResetEvent

Synchronized context

- SynchronizationAttribute, ContextBoundObject

Monitor class



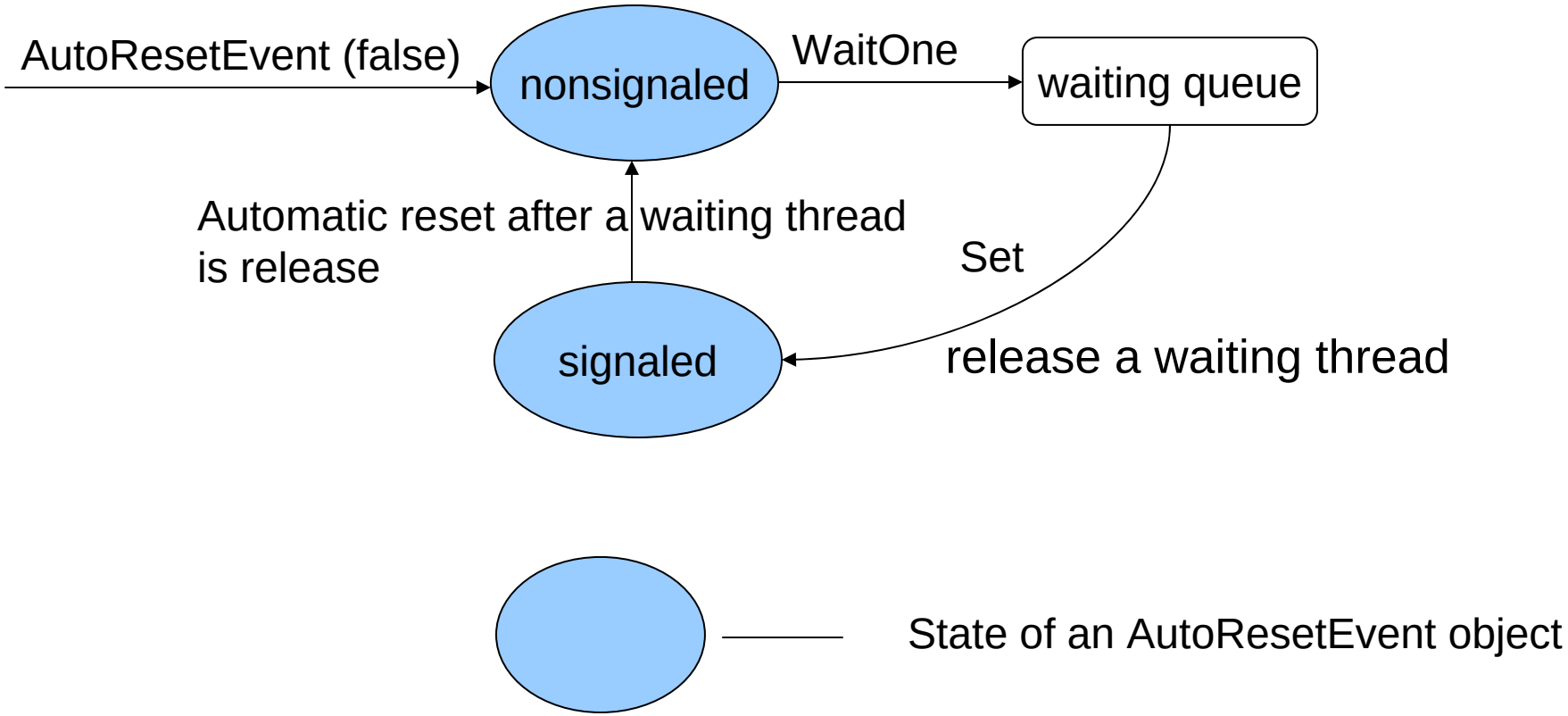
WaitHandle class

- A base class for all synchronization objects that allow multiply wait operations

- Derived classes
  - Mutex
  - AutoResetEvent
  - ManualResetEvent

- Define a signaling mechanism to take or release exclusive access to a shared resource

AutoResetEvent class



If set is called when no thread is waiting, the handle stays signaled as long as it takes until some thread to call WaitOne.

```
static AutoResetEvent ah = new AutoResetEvent(false);

static void Main(string[] args)
{
        for (int i = 1; i<=5; i++)
                new Thread(new ParameterizedThreadStart(Wait)).Start(i);
        ah.Set();
}

static void Wait(object no)
{
        int i = (int) no;
        Console.WriteLine("no. " + i + " is waiting.");
        ah.WaitOne();
        Console.WriteLine("no. " + i + " is notified");
}
```

Features of Wait and Pulse pattern

- Blocking conditions are implemented using custom fields
- Wait is always called within a statement that checks its blocking condition (itself within a lock statement)
- A single synchronization object is used for all Waits and Pulses and to protect access to all objects involved in all blocking conditions
- Locks are held only briefly

Wait and Pulse vs. Wait Handles

- Wait and Pulse pattern
  - Most flexible synchronization construct
  - Cannot work across multiply processes
  - Lock toggling
- Wait Handles
  - Work across multiply processes
  - Not lock toggleing, make induce implicit deadlock
  - More performance overhead under the condition that locks are uncontended
- Suggestion: Use wait / pulse except there are explicit waiting objects.

Serializing access to collections

- Most .NET classes are not thread-safe

- Collections like ArrayList, HashTable, Queue and Stack implement a method named Synchronized that returns a thread-safe version of the object passed to it

```
ArrayList list = new ArrayList();
ArrayList safeList = ArrayList.Synchronized (list);
...
//Thread A
safeList.Add(“Item A”);

//Thread B
safeList.Add(“Item b”);
```

However ....

Enumeration over a thread-safe collection is still unsafe

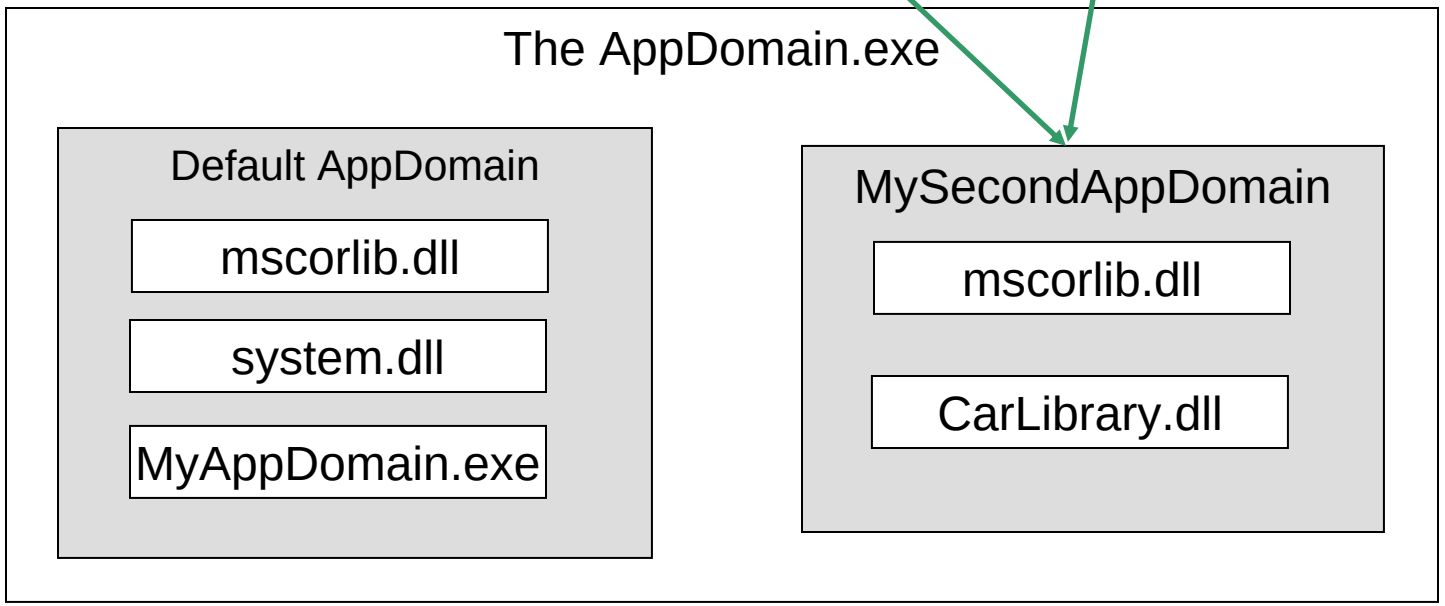
```
ArrayList list = new ArrayList();

...

//Thread A

lock (list.SyncRoot)
{
    list.Add("item 1");
    list.Add("item 2");
}

//Thread B
lock (list.SyncRoot)
{
    foreach (string office in list)
        ...
}
```

Automatic synchronization

- Application domains
  - Under .NET platform, a .NET assembly is hosted by a logical partition within a process termed an application domain (AppDomain)
- Object context boundaries
  - A given AppDomain is further subdivided into numerous context boundaries
  - Allow CLR to adjust the current method invocation to conform to the contextual settings of a given object
- Synchronized context
  - Allow you define a C# class type that requires automatic thread safty

Application domains


```
static void Main ()
{
    AppDomain anotherAD =
        AppDomain.CreateDomain("SecondAppDomain");
    anotherAD.Load("CarLibrary");
    ...
}
```



Using Synchronization attribute

- By deriving from ContextBoundObject and applying the Synchronization attribute, one instructs the CLR to apply locking automatically
- Can only be used to protect instance members
- Cannot be used to protect static type members

```
using System.Runtime.Remoting.Contexts;
[Synchronization]

public class AutoLock: ContextBoundObject
{
	public void Demo()
	{
		Console.Write(“Start...”);
		Thread.Sleep(1000);
		Console.WriteLine(“end”);
	}

	public void Test()
	{
		new Thread(Demo).Start();
		new Thread(Demo).Start();
		new Thread(Demo).Start();
	}

	public static void Main()
	{
		new AutoLock().Test();
	}
}
```

Atomicity and interlocked

- A statement is *atomic* if it executes as a single indivisible instruction
- In C#, a simple arithmetic operation is not atomic
- *Interlocked* class allow you to operate on a single point of data automatically with less overhead than with the locking mechanism

```
public void AddOne()
{
    int newVal = Interlocked.Increment (ref intVal);
}
```

ReaderWriterLock

- Defines a lock that supports single writers and multiple readers
- Used to synchronize access to a resource. At any given time, it allows either concurrent read access for multiple threads, or write access for a single thread

CLR ThreadPool

- Queue a method call for processing by a worker thread in the pool
- Register a Wait Handle along with a delegate to be executed when the Wait Handle is signed

- Benefit
  - Manages threads efficiently by minimizing the number of threads that must be created, started and stopped

- Note
  - All pooled threads are background threads

```
static AutoResetEvent ah = new AutoResetEvent(false);

public static void PrintNumber(object n)
{
	for (int i = 1; i <= (int)n; i++)
	{
		Console.WriteLine(i);
	}
}

public static void Main()
{
	ThreadPool.RegisterWaitForSingleObject(ah, Go, "hello", -1, true);
	Thread.Sleep(1000);
	ah.Set();

	WaitCallback workItem = new WaitCallback(PrintNumber);
	ThreadPool.QueueUserWorkItem(workItem, 10);
	Console.ReadLine();
}

public static void Go(object data, bool timeOut)
{
		Console.WriteLine("Started " + data);
}
```

Timer

The way of executing a method periodically

- Timer class
- TimerCallback delegate

```
static void PrintTime(Object state)
{
        Console.WriteLine("Time is: {0}", DateTime.Now.ToLongTimeString());
}

static void Main()
{
        TimerCallback timeCB = new TimerCallback(PrintTime);

        Timer t = new Timer(timeCB, "Hi", 0, 1000);
        Console.ReadLine();
}
```

Other things about threading

Synchronize access to entire method

```
[MethodImpl (MethodImpleOptions.Synchronize)]
public void foo()
{
    ...
}
```

Only one thread at a time can enter the method

Questions